Relatório

Trabalho Prático 1

Quicksort

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Paralela e Distribuída

Paradigmas de Computação Paralela

Carlos Rafael Cruz Antunes a67711

Nuno André da Silva Oliveira a67649

Resumo

O *quicksort* é um algoritmo que ordena um array dividindo-o em partes mais pequenas e ordenando-as. Este relatório serve para explicar a implementação sequencial e paralela no primeiro trabalho prático deste algoritmo. Serve também para estudar e analisar os resultados obtidos na medição de desempenho da versão sequencial e da versão paralela usando um diferente número de *cores* e um diferente número de tamanho de dados a ordenar. Usando estes dados podemos criar gráficos estatísticos e calcular o *speed up* de uma versão em relação à outra, e a escalabilidade do código produzido.

Algoritmo

A versão sequencial implementada foi baseada na apresentação deste trabalho: uma versão do quicksort recursiva que calcula o elemento a meio do *array* e vai trocando elementos que estejam à esquerda e que sejam maiores do que o elemento do meio por elementos da direita que sejam menores. Este algoritmo uma a técnica "divide and conquer", isto é, no fim da troca referida anteriormente, o *array* é dividido em dois, e é chamada a mesma função quicksort nos dois *arrays*.

A versão paralela foi implementada usando o algoritmo da versão sequencial dentro de uma região paralela, em que cada chamada recursiva é executada por uma *thread* diferente.

Ambiente de Teste

Os testes de desempenho foram executados no *cluster* da Universidade do Minho (*SeARCH*). Foi pedida uma máquina com 2 processadores com 12 *cores* (Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz) com hyper-threading. Totalizando em 48 *cores*, 24 destes virtuais. Esta máquina tem 32 *KBytes* de cache nível um para instruções e outros 32 *KBytes* de cache nível um para dados. Tem ainda 256 *KBytes* de cache nível dois e 30 *MBytes* de cache nível três.

Todas as medições foram executadas cinco vezes, e o valor utilizado para estatísticas foi o melhor valor nessas medições, porque é o que se aproxima mais do melhor caso. Foram medidos os tempos de execução para 1, 2, 4, 8, 16, 20, 28, 32, 40 e 48 *threads*.

Testes de Desempenho

Os testes de desempenho foram compilados com o comando "gcc -fopenmp quicksort.c". Devido a ocorrência de tempos em fila de espera demasiado grandes durante o fim-de-semana no *cluster* SeARCH, não foi possível realizar os testes de performance utilizando as *flags* de compilação -O2 e -O3 pelo que, para as versões paralelas todos os valores foram compilados sem *flags* de otimização.

Otimização do Código Sequencial

A versão do código sequencial foi compilada sem nenhuma *flag* de otimização tendo o melhor tempo sido 0,72 segundos, aproximadamente. Quando compilado com a *flag* -o3 o melhor tempo medido foi de 0,31 segundos, aproximadamente.

Cache Nível 1

Os testes neste nível de cache revelaram que para tamanhos do *array* muito baixos (5.000 elementos, 19.531 *KBytes* neste caso) o custo de paralelização excede o ganho, o que causa aumentos significativos no tempo de execução, semelhante ao que acontece na cache nível dois.

Cache Nível 2

Os testes neste nível de cache (*array* com 50.000 elementos, totalizando em 195 *KBytes*) à semelhança dos testes da cache nível um, não mostram grande melhoria no desempenho em relação à versão sequencial (uma *thread*). Isto deve-se ainda ao custo de paralelização ser elevado comparado ao custo de acesso a este nível de cache. Nos gráficos apresentados em baixo podemos verificar que ocorreu um ganho muito pouco significativo na versão paralelizada com duas e quatro *threads*, e uma perda de desempenho enorme nas restantes.

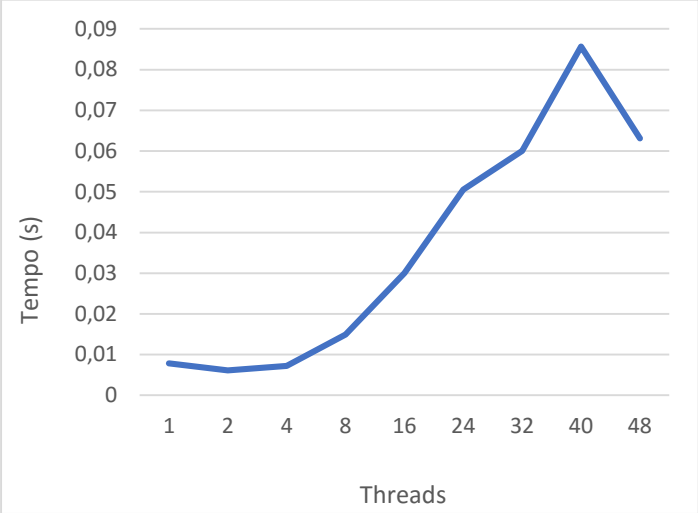

Gráfico 2 – Tempo de execução do algoritmo "*quicksort*" aplicado a um *array* de 50.000 elementos com diferentes níveis de paralelização.

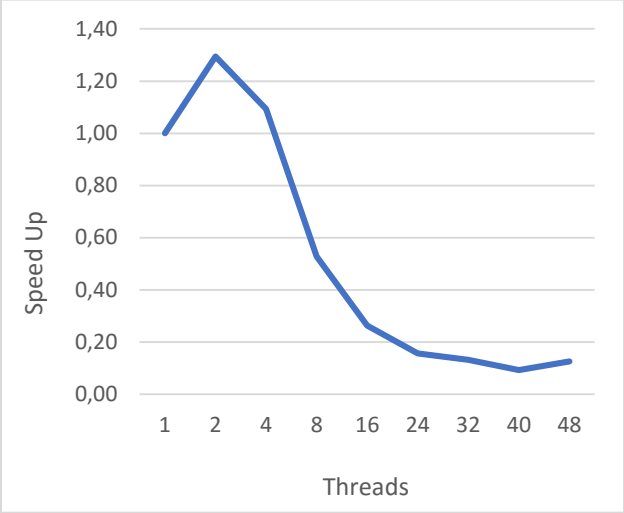

Gráfico 2 – *Speed Up* do algoritmo "*quicksort*" aplicado a um array de 50.000 elementos com diferentes níveis de paralelização.

Cache Nível 3

Os testes para este nível de cache (*array* com 5.000.000 elementos, totalizando em cerca de 19 *MBytes*) obtiveram resultados muito mais próximos aos esperados, atingindo um *speed up* próximo de quatro vezes, no entanto o programa isto só acontece quando o número de threads não excede oito. Quando isto acontece, verificou-se uma degradação no desempenho, embora ainda bastante superior à versão sequencial, como se pode verificar no *Gráfico 3* e no Gráfico 4, onde a linha a tracejado representa o tempo de execução ideal e o *speed up* ideal, respetivamente.

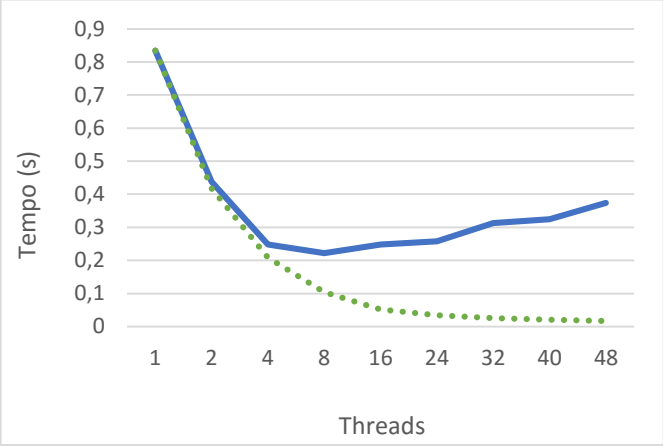

Gráfico 3 - Tempo de execução do algoritmo "*quicksort*" aplicado a um *array* de 5.000.000 elementos com diferentes níveis de paralelização.

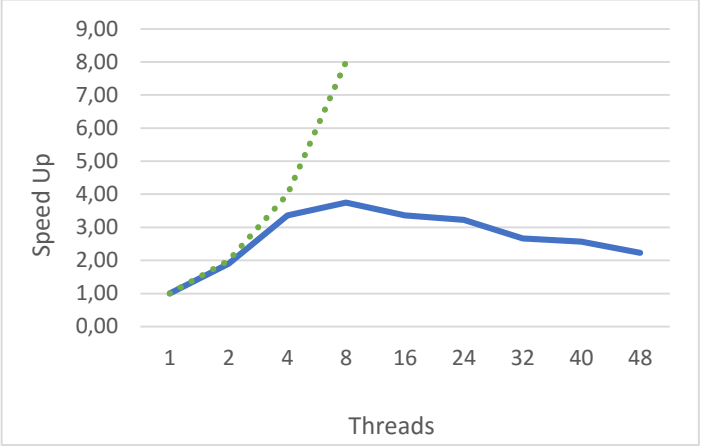

Gráfico 4 - *Speed Up* do algoritmo "*quicksort*" aplicado a um *array* de 5.000.000 elementos com diferentes níveis de paralelização.

Memória RAM

Para efetuar testes de desempenho no algoritmo quando este tem que usar a memória RAM, criou-se um *array* com vinte milhões de elementos, o que equivale a cerca de 76 *MBytes*. Verificou-se tempos de execução mais lentos do que obtidos nas memórias cache, como esperado, e verificou-se um aumento no desempenho do algoritmo, até um pico de 4.63 vezes mais rápido do que a versão sequencial, com oito *threads*. À semelhança dos resultados obtidos nas experiencias anteriores, quando o número de *threads* ultrapassa 4, o tempo de execução obtido afasta-se cada vez mais do tempo ideal esperado, como se pode verificar no Gráfico 5 (onde a linha a tracejado é o tempo ideal esperado) o que se reflete no *speed up* (Gráfico 6).

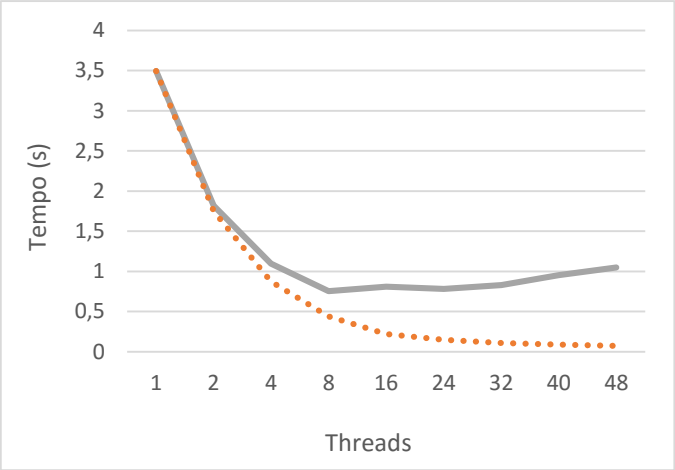

Gráfico 5 - Tempo de execução do algoritmo "*quicksort*" aplicado a um *array* de 20.000.000 elementos com diferentes níveis de paralelização.

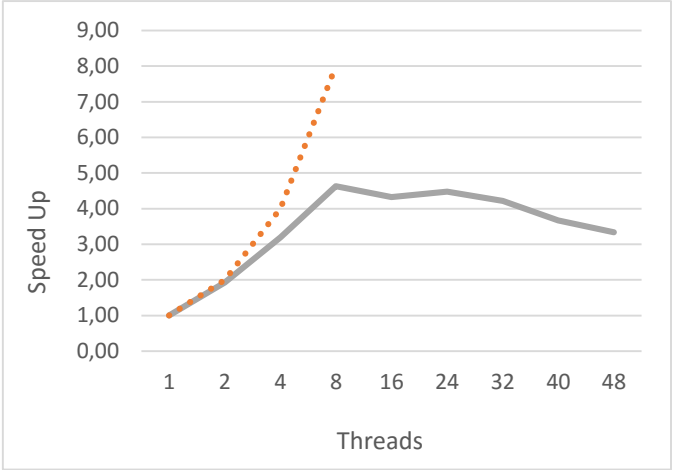

Gráfico 6 - *Speed Up* do algoritmo "*quicksort*" aplicado a um *array* de 20.000.000 elementos com diferentes níveis de paralelização.

Análise de Resultados

Através dos resultados obtidos podemos concluir que o programa escala mal na maioria dos casos, obtendo um *speed up* máximo próximo de quatro mesmo quando utilizado um número de *threads* bastante superiores a esse numero. Podemos também concluir que os resultados obtidos são melhores quando é utilizado um *array* grande, pois caso contrário, o *overhead* introduzido pela paralelização é superior aos seus ganhos. Uma das razões para a má paralelização do programa é o facto de as primeiras iterações da recursividade do algoritmo não utilizarem todas as *threads* disponíveis.